CONTEXT-CORRUPTING CONTEXT SWITCHING

BACKGROUND

[0001] An operating system (OS) can be run as application on a computer using a virtual machine. The OS that is run as the application is referred to as the “guest OS,” and the underlying OS is referred to as the “host OS.” For example, a Windows-based OS can be run as an application on top of a host Linux OS. Running the Windows-based OS as an application allows programs written for a Windows environment to be run on top of the host Linux OS.

[0002] The guest OS usually has a different “context” than the host OS. A context embodies the accessible state of the computer’s central processing unit (CPU). The context includes in particular the values of CPU registers. It does not include resources that the CPU would not allow currently executing code to access.

[0003] The values of certain registers may need to be different to execute the guest OS than to execute the host OS. The virtual machine can allow the guest OS and the host OS to run concurrently by properly restoring the context of the guest OS whenever control is transferred from the host OS to the guest OS, and by similarly restoring the context of the host OS when control is transferred from the guest OS to the host OS. This process is called “context switching”, and is similar in principle to switching the context between applications in time-shared environments. Most CPU architectures offer built-in support for switching context between applications, making it possible to completely save and restore the context of any application. This support often includes saving selected CPU resources when an interruption occurs, and reserving certain registers for use only by the OS.

[0004] Certain CPU architectures, such as IA-64, have prohibitions against, and difficulties with, completely saving and restoring the entire context of the guest OS or host OS. Specifically, certain registers might be corrupted by the context switching process. If the entire context cannot be saved, the guest OS or host OS might behave incorrectly and crash.

SUMMARY

[0005] According to one aspect of the present invention, context is switched on a processor by saving the context under software control using an inconsequential register as temporary storage; and preventing the processor from changing the context while the context is being saved. Other aspects and advantages of the present invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, illustrating by way of example the principles of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] Figure 1 is an illustration of hardware and software components of a computer.

[0007] Figure 2 is a flowchart of steps performed during a context switch.

[0008] Figure 3 is an illustration of context switches between a host operating system, a virtual machine, and a guest operating system.

DETAILED DESCRIPTION

[0009] As shown in the drawings for purposes of illustration, the present invention is embodied in a computer including a CPU and software components including a host OS, and a virtual machine (VM) application. The host OS creates an environment in which applications (such as the VM application) can run. The VM application, when executed, changes the environment created by the host OS to a different environment. The virtual machine enables execution of programs that are written to run in that different environment.

[0010] One well-known type of virtual machine is a Java virtual machine. A program such as a Java program would see only the Java environment, even though the host OS created a different environment (e.g., a Windows-based or Unix-based environment). The Java virtual machine "emulates" a CPU by translating Java bytecode into native code, and running the native code on the CPU. Although the present invention may be applied to such a virtual machine, it is not so limited. The present invention may be applied to a virtual machine that does not perform CPU emulation. An advantage of a VM that does not perform CPU emulation is that programs are mostly run "native" and, hence, at full speed. Such a virtual machine can be used to run applications for one operating system on top of a different operating system for the same processor, for instance running Windows applications in a Windows environment created by a virtual machine running on Linux. This is achieved by running the guest OS inside the virtual machine.

[0011] The context switching will now be described in connection with a simple CPU having three application-accessible registers I, J and K, and a privileged register X (later, the context switching will be expanded to cover the IA-64 processor). The CPU controls access to the privileged register X based on the Privilege Level (PL) at which code executes. PL0 is the highest privilege, and is used for operating system code. PL3 is the lowest privilege, and is used for application code. Code running at PL0 is called privileged code.

[0012] A write operation by the CPU may involve at least two registers. One register stores a target address and at least one other register stores values to be written to the target address.

[0013] Reference is made to Figure 1, which shows different hardware and software components of a computer. The hardware components include a CPU 110 and random access memory (RAM) 112. A host OS 210 runs at PL0 and, therefore has full access to the CPU 110 of the computer, and in particular to the privileged register X. For correct execution of applications, the host OS 210 is responsible for saving and restoring any application-visible processor state not automatically saved by the CPU 110. This includes the values of the application-accessible registers I, J and K. However, the host OS 210 in general, and interrupt handling routines in particular, are free to use the privileged register X.

[0014] Host and guest applications 218 and 220 normally run at a lower privilege, such as PL3. The host OS 210 does not allow the host and guest applications 218 and 220 to access the privileged register X.

[0015] The host OS 210 runs PL0 code and uses the privileged register X as temporary storage to switch context to and from host and guest applications 218 and 220. Context switching between applications can occur at any time, for instance in response to an interrupt. Consider the following example of switching between applications A and B. When an interrupt occurs, the host OS 210 performs the following:

```
        X = address of application A context
        store I, J, K to X
        X = address of application B context
[1]     load I, J, K from X
```

The address stored in register X is typically an address in RAM 112. Later, the host OS 210 restores the context of application A with the same process, which ends with:

```
        X = address of application A context
        load I, J, K from X
```

Had application A stored a value in the privileged register X, that value would have been lost during the context switching. However, the host OS 210 prevents application A from using the privileged register; therefore, the context of application A (that is, the values of the application-accessible registers I, J and K) is identical before and after the interruption.

[0016] Thus, application context switching destroys the value of the privileged register X. However, since the host and guest applications 218 and 220 cannot access the privileged register X, they are not affected by context switching.

[0017] Destroying the value of the privileged register X creates a problem if an attempt is made to switch context between the host OS 210 and a guest OS 214. The context of the host OS 210 includes the privileged register X as well as the application-specific registers I, J and K. During the context switch, the privileged register X would be used to specify the target address of the other registers I, J and K. However, once the privileged register X is modified, the context of the host OS 210 is modified and cannot be restored in its entirety. Corrupting the privileged register X would change the behavior of the host OS 210 upon restoration and cause it to crash

[0018] A virtual machine application 212 and driver 216 are used to avoid this problem. When executed, the virtual machine application 212 changes the environment created by the host OS to a different environment, that of a virtual machine (VM). The driver 216 runs in the most-privileged mode PL0. The virtual machine application 212, which initially runs at PL3, invokes the driver 216 to take control of the CPU 110. The VM application 212 may invoke the driver 216 by using a host API such as "ioctl" (for a UNIX-based OS) or "DeviceControl" (for a Windows-based OS). The driver 216 allows the virtual machine application 212 to access the privileged register X and therefore use it as temporary storage.

[0019] The guest OS 214 runs in the VM at PL1 instead of PL0. Therefore, the guest OS 214 has only partial access to the CPU 110 and can only access the privileged register X through the VM. One of the roles of the VM is to simulate accesses to the privileged register X made by the guest OS 214, as if the guest OS 214 were running normally at PL0.

[0020] Figure 2 illustrates the operation of the VM application 212 in connection with a context switch between the host OS 210 and the guest OS 214. Before the context is switched from the host OS 210 to the guest OS 214, the VM application 212 temporarily disables the interrupts (310) to prevent the privileged register X from being corrupted during the context switching. For instance, disabling the interrupts during the context switch would prevent interrupt handlers from changing the value of the privileged register X.

[0021] Next, the virtual machine application 212 saves the context of the host OS (312) using an inconsequential register as temporary storage. The point at which the host OS context is saved is under software control (synchronous) rather than under control of an external event such as a timer (asynchronous). This point will therefore be referred to as a predetermined interruption point (PIP). An inconsequential register is a register that is not used by the host OS 210 at the PIP.

[0022] Since the context of the host OS 210 is saved at the PIP, it is known which registers the host OS 210 uses and which registers the host OS 210 does not use. Therefore, the inconsequential register(s) at the PIP can be identified. The inconsequential registers can be corrupted by the virtual machine application 212 without affecting the host OS 210. In particular, the context switching process can use the inconsequential registers as a temporary storage in lieu of the privileged registers.

[0023] For example, if register I is an inconsequential register at the PIP, the context of the host OS 210 may be saved as follows.

```
      I = address of OS A context
[2]   store X, J, K to I
      switch to  context of virtual machine
```

Switching the context to the VM (314) corrupts the registers I, J, K and X with random values. However, the contents of registers J, K and X were stored at the address indicated by register I. Since I is an inconsequential register, the entire meaningful context of the host OS 210 is saved.

[0024] Once the context has been switched to that of the virtual machine, the guest OS 214 may be run as an application (316), under control of the virtual machine. At that point, the virtual machine has taken over CPU ownership normally assumed by the host OS 210. The virtual machine saves the context of the guest OS 214 just like the host OS 210 would for a host application 218.

[0025] The VM application 212 may relinquish control of the CPU 110 to the host OS 210 at any time, for instance in response to a timer interrupt that indicates the end of the time slice allotted to the VM application 212. When a switch is made from the VM context back to the host OS 210, the virtual machine application 212 saves the context of the guest OS (318). Since the guest OS 214 runs unprivileged, saving the guest OS context is similar to saving the context of an application, and can make use of the same kind of reserved resources (except that they are at this point reserved to the virtual machine rather than to the host OS).

[0026] On the other hand, restoring the host OS context (320) requires restoration of the privileged register X. The virtual machine application 212 cannot use the privileged register, but it can (and does) use the inconsequential register(s) as temporary storage.

[0027] Continuing with the example above, the context of the host OS 210 may be restored as follows.

```
[3]    I = address of OS A context
       load X, J, K from I
```

Thus the value of the privileged register X is now correctly preserved.

[0028] Later, control may be returned to a known point [4], at which execution of the virtual machine application 212 or driver 216 resumes, now under control of the host OS 210. If the value of register I was saved separately by a software convention between points [2] and [4], the register I can be used to pass additional information back to the virtual machine application.

[0029] Figure 3 provides an example of a series of context switches. The VM application 212 is started at PL3 and calls the driver 216, which is running at PL0. The driver 216 allows the VM application 212 to take control of the CPU 110. The virtual machine is activated at point A. Between points A and B, the context of the host OS 210 is preserved in RAM 112 or other storage by the virtual machine. At point B the virtual machine saves the context of the guest OS 214, the context of the host OS 210 is restored, the virtual machine is exited and control of the CPU 110 is returned to the host OS 210.

[0030] The simple CPU 110 was used to facilitate an understanding of the context switching according to the present invention. The context switching may be applied to a processor such as the IA-64 processor. The IA-64 processor includes general registers (GR) containing integer values, branch registers (BR) containing the address of code to branch to, control registers (CR) with specialized roles, reserved for use by operating systems, and banked general registers (BGR) which only an OS can use. By software convention, some of these registers are caller-save (scratch) registers. The IA-64 processor controls access to the control registers (CR) and the banked general registers (BGR) based on the PL. Therefore, the control registers (CR) and the banked general registers (BGR) are privileged registers. The general registers (GR) and the branch registers (BR) are application-accessible registers.

[0031] For correct execution of applications, the host OS is responsible for saving and restoring any application-visible processor state not automatically saved by the IA-64 processor. This includes the values of the general and branch registers (GR and BR). However, the host OS in general, and interrupt handling routines in particular, are free to use the control registers (CR) and the banked general registers (BGR)

[0032] Caller-save registers, also called scratch registers, are of particular interest. By software convention, these registers are modified arbitrarily across a call. Thus, code is not allowed to use their value immediately after a call. If the PIP is made to be the instruction following a call, then the virtual machine is allowed to corrupt all scratch registers. There is no restriction in making the PIP immediately follow a call to a context saving routine. This technique makes all scratch registers available to the VM application.

[0033] Branch registers also present a special interest. It may be necessary or practical for the virtual machine to corrupt at least one branch register in the host OS context. If the virtual machine returns control to the host OS at an address that is computed during runtime, an indirect branch might be used to vector to the target address. This target address is computed during runtime to select one of multiple possible choices (e.g., different interrupt handlers). The indirect branch can also be used to avoid branch distance limitations that exist on some CPUs such as the Itanium implementation of the

IA-64 architecture. On most RISC CPUs and on IA-64, this indirect branch will corrupt at least one register. Any branch register that is ignored at the PIP can be used. The return branch register (branch register 0 on IA-64) is a scratch register, since it is corrupted by the return of the call. Thus, if the PIP is an instruction following a call, it is possible for the virtual machine to corrupt this branch register as part of its execution.

[0034] The present invention is not limited to resuming execution precisely at the PIP. Execution may resume at any point different than the PIP, as long as the selected resumption point does not use the registers that the virtual machine may have corrupted. The virtual machine may use this possibility to dynamically select a different resumption point for different return paths to the host OS.

[0035] The present invention is not limited to the IA-64 architecture and may be applied to any other CPU architecture having registers reserved for use by the operating system and limitations on saving and restoring all registers that an operating system can access.

[0036] The present invention is not even limited to CPU architectures having registers reserved for use by the operating system and limitations on saving and restoring all registers that an operating system can access. It may be applied to CPU architectures that make all registers available, without restriction. All registers could be correctly saved and restored, but saving or restoring only part of them may still be more efficient. Efficiency may be increased by saving and restoring a smaller number of registers because this frees up additional registers for the interrupting code, or both. Thus, the present invention may allow more efficient management of the registers.

[0037] The manner in which the CPU is prevented from changing the context while the context is being saved is a matter of design choice. The manner described above (running a driver at PL0 to gain access to the privileged registers and disable the interrupts) is but one example.

[0038] Although the present invention was described in connection with a virtual machine application, it is not so limited. The virtual machine application is but one example of an application that needs to inject code without disturbing the surrounding context. The present invention may be used by any other software application that needs to execute code without disturbing a surrounding context. Other examples of such applications include debuggers, performance measurement tools, profiling tools, etc.

[0039] The present invention is not limited to the specific embodiments described and illustrated above. Instead, the present invention is construed according to the claims that follow.